Como Implementar um Emulador

Lucas Pascotti Valem

lucaspascottivalem@gmail.com
Universidade Estadual Paulista (UNESP)
Rio Claro, São Paulo, Brasil

20 de Outubro de 2016

unesp

SECCOMP

Roteiro

Introdução

Definição

Emulador é um software que **reproduz as funções de um determinado ambiente**, a fim de permitir a execução de outros softwares sobre ele.

Exemplo

Dado um programa de computador (binário) de uma arquitetura/sistema A. Se desejarmos executar esse programa em uma arquitetura/sistema B a partir do mesmo binário, precisamos que um emulador nos ajude nessa tarefa.

Introdução

Um **emulador** pode utilizar de **três diferentes estratégias** para emular um hardware.

Formas de Emulação

- **Intepretação**
- **Recompilação estática**
- **Recompilação dinâmica**

A combinação desses métodos é possível, mas o **mais comum** é que emuladores sejam apenas **interpretadores**.

Introdução

Lembre-se, um **emulador é completamente diferente de um simulador**!

Emulador

Reproduz o comportamento do hardware em outra plataforma.

Simulador

Reproduz o comportamento do software.

Introdução

Curisiodade

O **primeiro emulador** foi criado em **1964** por Larry Moss, na época funcionário da **IBM**, consistindo em um Software que fazia com que os programas criados para o 7070 mainframe rodassem na mais nova linha de computadores da IBM, os System/360.

Consoles

Quando **emular** está associado a um hardware, por exemplo **vídeo games**, o **emulador faz o trabalho do console**, que por sua vez necessita de ROMs que é uma cópia do jogo (software).

Conceitos

Conceitos:

- **ROM (Read Only Memory)**: Armazenamento de um software em forma binária;
- **Registradores**: Menor unidade de memória (armazena um valor de n bits);
- **Instruções**: São as operações mais básicas de uma CPU, possuindo mnemônicos em linguagem de montagem.
- **Assembler**: Converte o código de linguagem de montagem para binário;

Exemplo

ADD V1, V2
ADD é uma instrução.
V1 e V2 são registradores.

Como implementar?

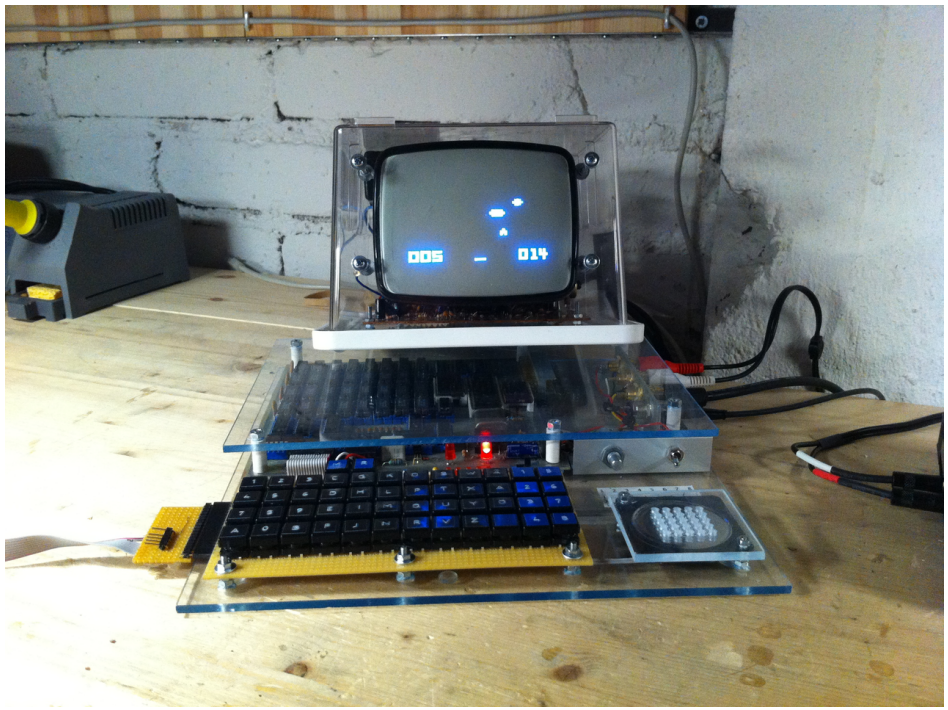
Para **implementar um emulador**, precisamos ter as **informações do hardware** que queremos emular:

- Tamanho e distribuição da **Memória**
- Número, tamanho e função dos **Registradores**
- Tamanho da **ROM**
- **Conjunto de instruções** do processador
- Funcionamento da **placa gráfica** (se houver)

Entre outras informações que são quase sempre obtidas por **engenharia de reversa** feita no **hardware** original e compiladas em uma **documentação**.

Escolha do "Hardware"

Iremos implementar um **emulador** de **Chip-8**, uma linguagem interpretada desenvolvida por Joseph Weisbecker em meados de **1970** e utilizada em diversos **videogames** da época.



Chip-8

Abaixo uma foto do **Telmac 1800** que funcionava sobre o **Chip8**.

Informações Gerais

A **implementação** do emulador será realizada em **C/C++** tendo como única **depêndencia** o **SFML** (gráficos, input/output, etc).

Documentação, slides e códigos em:

https://github.com/lucasPV/Chip8Seccomp

Download do SFML:

http://www.sfml-dev.org/download/sfml/2.4.0/

Nessa apresentação, cada um dos detalhes da documentação do Chip-8 será comentada e sua respectiva implementação será mostrada. Ao término da apresentação temos uma atividade!

Sistema Operacional

A escolha do sistema operacional fica por conta de vocês! Apenas certifiquem-se que possuem um compilador de C/C++ e o sistema seja compatível com o SFML. No entanto, recomenda-se fortemente um sistema GNU/Linux para a execução das atividades.

Começando do zero

Vamos **começar do zero**!

chip8.cpp

```
#include <stdio.h>

int main(int argc, char* argv[]) {

        return 0;
}
```

Definição de tipos

Vamos definir dois tipos básicos para facilitar a legibilidade e escrita do código:

- byte = 8 bits
- word = 16 bits

chip8.cpp

```
//types
typedef unsigned char  byte;
typedef unsigned short word;
```

Memória

A memória é de 4KB, sendo:

- 512B reservados para fontes e interpretador
- 3.5KB para a ROM

chip8.cpp

```
//main memory
const int memSize  = 0xFFF; //4KB
const int fontSize = 0x200; //512B are used to store fonts
byte memory[memSize];       //stores fonts and the ROM
```

Vídeo

O display é monocromático de 64x32, vamos representá-lo como uma matriz de booleanos (0 ou 1).

chip8.cpp

```
//graphical memory
const int displayWidth  = 64;
const int displayHeight = 32;
bool display[displayWidth*displayHeight];
```

Teclado

O Chip-8 possui um teclado de 16 teclas. Vamos representá-lo como um vetor de booleanos, sendo:

- 0 = não pressionado
- 1 = pressionado

chip8.cpp

```
//keyboard
bool key[16];
```

Registradores

De 8 bits (1 byte):

- Registradores Gerais, de V[0] a V[F]:
  - V0-VE: Propósito Geral
  - VF: Carry, Borrow e Detecção de Colisões
- delayTimer, soundTimer: Realizam contagem para o timer

De 16 bits (1 word):

- I: Guarda endereços temporários
- PC: Contador de programa, aponta para próxima instrução
- SP: Aponta para o topo da pilha de chamada (*call stack*)

chip8.cpp

```
//registers
byte V[0xF]; //V0-VE: General Purpose; VF: Carry, Borrow and Collision Detection
word I, PC, SP; //I: Index; PC: Program Counter; SP: Stack Pointer
byte delayTimer, soundTimer;  //timer registers that count at 60 Hz
```

Pilha de Chamadas

A pilha de chamadas (*stack*) possui 16 níveis e armazena palavras de 16 bits (1 word).

chip8.cpp

```
const int stackLevels = 16;
word stack[stackLevels];
```

Fontset

O Chip-8 possui um conjunto de fontes gravado em memória, na documentação pode ser encontrado o valor hexadecimal de cada uma das fontes. Abaixo alguns exemplos:

"0"	Binary	Hex
****	11110000	0xF0
* *	10010000	0x90
* *	10010000	0x90
* *	10010000	0x90
****	11110000	0xF0

"1"	Binary	Hex
*	00100000	0x20
**	01100000	0x60
*	00100000	0x20
*	00100000	0x20
***	01110000	0x70

"2"	Binary	Hex
****	11110000	0xF0
*	00010000	0x10
****	11110000	0xF0
*	10000000	0x80
****	11110000	0xF0

"3"	Binary	Hex
****	11110000	0xF0
*	00010000	0x10
****	11110000	0xF0
*	00010000	0x10
****	11110000	0xF0

Fontset

A partir da documentação podemos definir o fontset em nosso código:

chip8.cpp

```
//fontset
const byte fontset[80] = {
    0xF0, 0x90, 0x90, 0x90, 0xF0, //0
    0x20, 0x60, 0x20, 0x20, 0x70, //1
    0xF0, 0x10, 0xF0, 0x80, 0xF0, //2
    0xF0, 0x10, 0xF0, 0x10, 0xF0, //3
    0x90, 0x90, 0xF0, 0x10, 0x10, //4
    0xF0, 0x80, 0xF0, 0x10, 0xF0, //5
    0xF0, 0x80, 0xF0, 0x90, 0xF0, //6
    0xF0, 0x10, 0x20, 0x40, 0x40, //7
    0xF0, 0x90, 0xF0, 0x90, 0xF0, //8
    0xF0, 0x90, 0xF0, 0x10, 0xF0, //9
    0xF0, 0x90, 0xF0, 0x90, 0x90, //A
    0xE0, 0x90, 0xE0, 0x90, 0xE0, //B
    0xF0, 0x80, 0x80, 0x80, 0xF0, //C
    0xE0, 0x90, 0x90, 0x90, 0xE0, //D
    0xF0, 0x80, 0xF0, 0x80, 0xF0, //E
    0xF0, 0x80, 0xF0, 0x80, 0x80  //F
};
```

Função de Startup

A função de *startup* é responsável em emular o *boot* do *Chip8*, inicializando as estruturas:

chip8.cpp

```
void startup() {
    srand(time(NULL));  //seed random numbers

    PC = 0x200;         //ROM to be loaded at memory location 0x200
    I  = 0;             //reset index register
    SP = 0;             //reset stack pointer
    delayTimer = 0;     //reset delay timer
    soundTimer = 0;     //reset sound timer

    //clear registers V0-VF
    for (int i = 0; i < 0xF; i++) {
        V[i] = 0;
    }

    //clear stack
    for (int i = 0; i < stackLevels; i++) {
        stack[i] = 0;
    }
```

Função de Startup

chip8.cpp

```
    //load fontset into memory
    for (int i = 0; i < 80; i++) {
        memory[i] = fontset[i];
    }

    //clear graphical memory
    for (int i = 0; i < displayWidth*displayHeight; i++) {
        display[i] = 0;
    }

    //clear keyboard
    for (int i = 0; i < 0xF; i++) {
        key[i] = 0;
    }
}
```

Carregando a ROM

A função *loadROM* carrega a ROM na memória:

chip8.cpp

```
void loadROM(const char* filename) {
    //open file
    FILE* file = fopen(filename, "rb");
    if (file == NULL) {
        printf("Couldn't open ROM: %s\n", filename);
        exit(1);
    }
    //get file size
    fseek(file, 0, SEEK_END);
    long size = ftell(file);
    rewind(file);
    if (size > (memSize-fontSize)) {
        printf("ROM too big for Chip8 Memory (more than 3.5KB)!\n");
        exit(1);
    }
    //read file into memory
    fread(memory + fontSize, sizeof(byte), size, file);
    //close
    fclose(file);
}
```

Estruturando o Main

Até o momento, o main deve ser estruturado da seguinte forma:

chip8.cpp

```
//includes

//structures definition

int main(int argc, char* argv[]) {
    if (argc < 2) {
        printf("ROM␣not␣specified!\n");
        exit(1);
    }

    startup();

    loadROM(argv[1]);

    while (1)
        //emulateCycle();

    return 0;
}
```

Ciclo de Emulação

É a principal função do emulador, responsável pelas seguintes etapas em ordem:

- Busca de Instrução
- Atualiza o PC
- Executa a Instrução
- Atualiza os Timers

Sendo executada 60 vezes por segundo (futuramente SFML nos assegurará esse tempo).

chip8.cpp

```
void emulateCycle() {

}
```

Ciclo de Emulação: Busca de Instrução

Cada instrução é de 16 bits, por isso precisamos concatenar a parte mais baixa a mais alta da memória a partir de PC:

chip8.cpp

```
//fetch instruction
word instr = ((memory[PC] << 8) | memory[PC + 1]);
```

Ciclo de Emulação: Atualizar PC

Como PC é de 16 bits, incrementamos em duas unidades

chip8.cpp

```
//update PC
PC += 2;
```

Ciclo de Emulação: Instruções

A partir das instruções, vamos extrair os campos de interesse:

chip8.cpp

```
//extract bit-fields from the opcode
byte p   = ((instr & 0xF000) >> 12);
byte x   = ((instr & 0x0F00) >> 8);
byte y   = ((instr & 0x00F0) >> 4);
byte kk  = (instr & 0x00FF);
word nnn = (instr & 0x0FFF);
byte n   = (instr & 0x000F);
```

Ciclo de Emulação: Instruções

Pela documentação, vemos que o **Chip-8** possui **35 instruções** a serem implementadas. Para isso, **implementamos um switch a fim de selecionar a instrução a ser executada**. Serão explicadas as principais instruções, a **implementação completa** estará disponível no **github** após o **término da atividade**.

Instrução 00E0 - CLS

Simplesmente limpa a tela, podemos fazer isso limpando a memória de vídeo.

chip8.cpp

```
void clearDisplay() {
    for (int i = 0; i < displayWidth*displayHeight; i++) {
        display[i] = 0;
    }
}
```

Instrução 1nnn - JP

PC passa a apontar para o endereço nnn.

chip8.cpp

```
PC = nnn
```

Instrução 2nnn - CALL addr

Chama a subrotina em nnn.

chip8.cpp

```
stack[SP] = PC;
SP++;
PC = nnn;
```

Instrução 3xkk - SE Vx, byte

Pula a próxima instrução se Vx = kk.

chip8.cpp

```
if (V[x] == kk) {
    PC += 2;
}
```

Instrução 8xy0 - LD Vx, Vy

Carrega em Vx o valor de Vy

chip8.cpp

```
V[x] = V[y];
```

Instrução 8xy4 - ADD Vx, Vy

Vx armazena a soma de Vx e Vy e Vf guarda o carry.

chip8.cpp

```
tmp = V[x] + V[y];
V[0xF] = (tmp >> 8);
V[x] = tmp;
```

Instrução 8xy5 - SUB Vx, Vy

Vx armazena a subtração de Vx e Vy e Vf guarda o borrow.

chip8.cpp

```
tmp = V[x] - V[y];
V[0xF] = !(tmp >> 8);
V[x] = tmp;
```

Instrução Cxkk - RND Vx, byte

Gera um número aleatório entre 0 e 255 e faz o AND desse número com o valor kk. O resultado é armazenado em Vx.

chip8.cpp

```
V[x] = ( rand() % 0xF & kk );
```

Instrução Dxyn - DRW Vx, Vy, nibble

Instrução responsável pelo desenho.

chip8.cpp

```
void draw(byte x, byte y, byte height) {
    word pixel;

    x = x%64;
    y = y%32;

    V[0xF] = 0;
    for (int yline = 0; yline < height; yline++) {
        pixel = memory[I + yline];
        for (int xline = 0; xline < 8; xline++) {
            if ((pixel & (0x80 >> xline)) != 0) {
                if (display[(x + xline + ((y + yline) * 64))] == 1) {
                    V[0xF] = 1;
                }
                display[x + xline + ((y + yline) * 64)] ^= 1;
            }
        }
    }
}
```

Ciclo de Emulação: Atualizar os Timers

Os timers são decrementados enquanto são maiores que 0.

chip8.cpp

```
if (delayTimer > 0)
    delayTimer--;
if (soundTimer > 0) {
    //sound.play();
    soundTimer--;
}
```

Atividade

No final da implementação das intruções, o SFML será utilizado para três funções específicas:

- Gráficos
- Controles
- Som

Atividade

Agora vamos realizar uma atividade! Implementar as instruções restantes!
Acessem: **https://github.com/lucasPV/Chip8Seccomp**